# 集合框架

## 集合综述

Scala的集合有三大类：序列Seq、集Set、映射Map，所有的集合都扩展自Iterable特质

在Scala中集合有可变（mutable）和不可变（immutable）两种类型(两个不同的包)，

包的全局路径：scala.collection.immutable 默认使用的是该包中的集合

如果想使用可变的集合，就需要导包 scala.collection.mutable

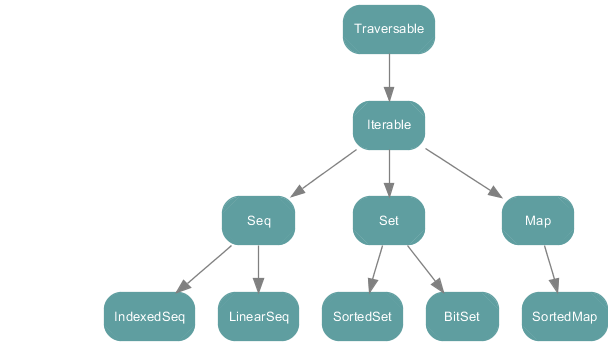
immutable类型的集合初始化后就不能改变了（注意与val修饰的变量进行区别）

使用val这个关键字修饰一个变量（相当于java中的final），那么就意味着该变量的引用不可变，该引用中的内容是不是可变，取决于这个引用指向的集合的类型

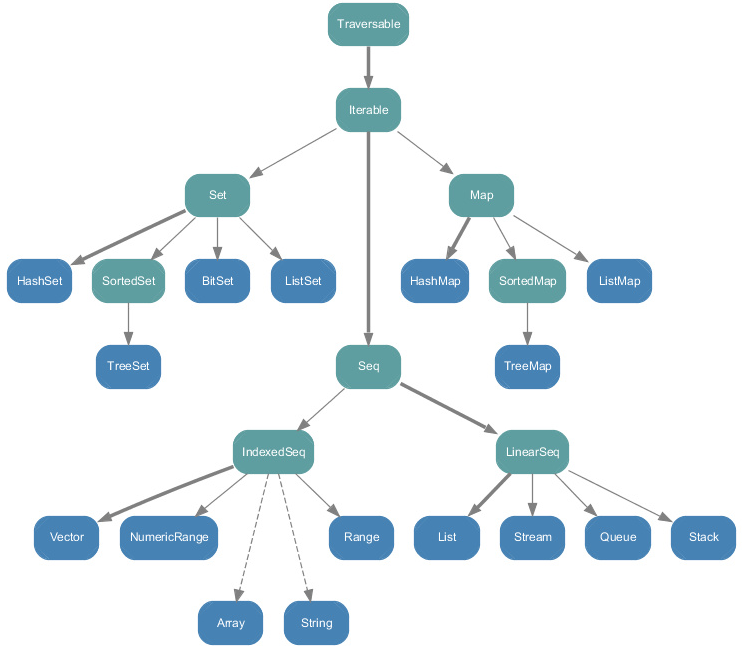
val 集合 ，该集合中的内容是否可变，取决于集合的特性。

## 集合框架整体架构（了解）

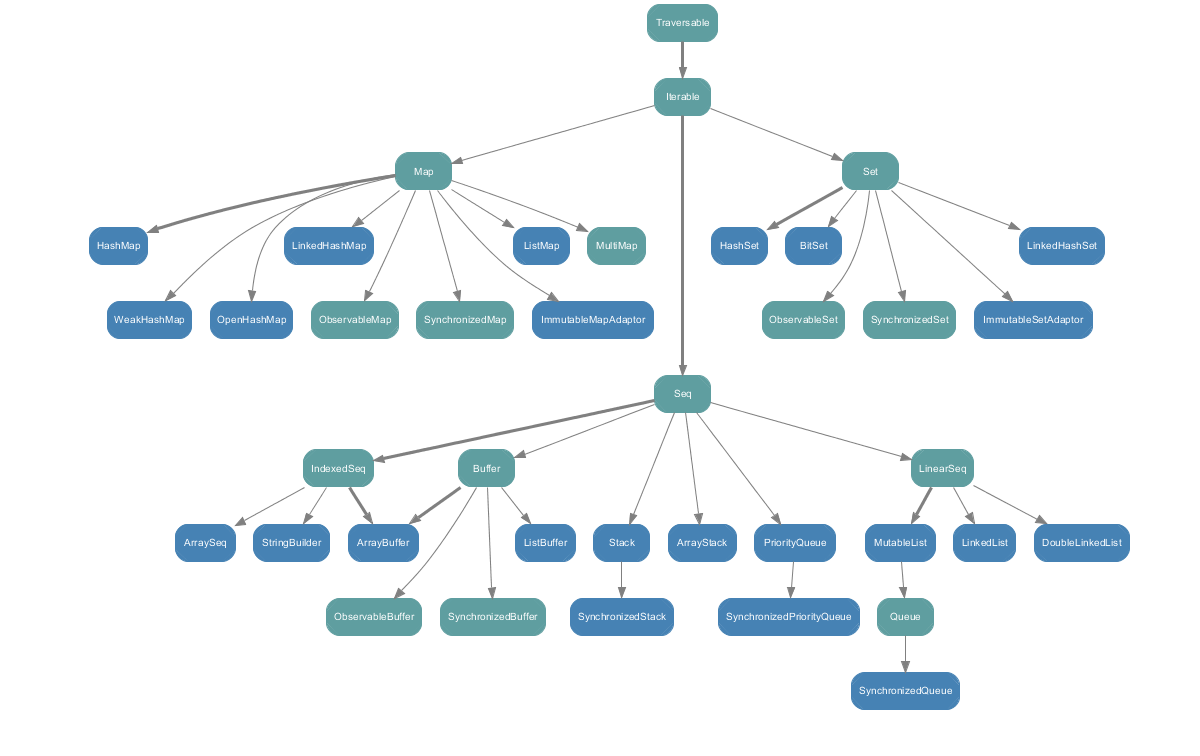
顶级抽象类或父类：



不可变集合（scala.collection.immutable）：



可变集合(scala.collection.mutable)：



# 数组

定长数组，长度不可变，但是内容可变。 默认是用的是定长数组

可变数组，长度，内容都是可变的。

## 定长数组和变长数组

Nothing是所有类型的子类型，表示程序出错了。

Null ,所有引用类型的子类型，只有一个值，就是null

定长数组，是不可变的，**长度不可变，内容可变**，默认类型

变长数组，可变的，长度可变，内容可变

创建定长数组：

指定数组类型，并赋值，类型可省略，编译器会自动推导。

**val** arr1 = *Array*[Int](1,2,3,45)

通过new关键字创建的数组，**必须指定数组的类型**和长度。  
**val** arr2 = **new** Array[Int](10)

取值赋值，下标从0开始：

arr1(0)

arr1(0) =100

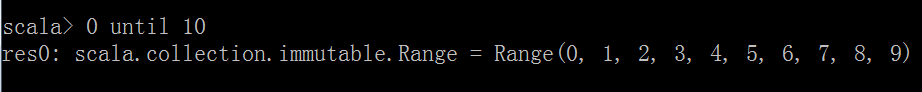
|  |
| --- |
| **object** ArrayTest {   **def** main(args: Array[String]) {   *//初始化一个长度为8的定长数组，其所有元素均为0* **val** arr1 = **new** Array[Int](8)  *//直接打印定长数组，内容为数组的hashcode值  println*(arr1)  *//将数组转换成数组缓冲，就可以看到原数组中的内容了  //toBuffer会将数组转换长数组缓冲  println*(arr1.toBuffer)   *//注意：如果不是new，相当于调用了数组的apply方法，直接为数组赋值  //初始化一个长度为1的定长数组* **val** arr2 = *Array*[Int](10)  *println*(arr2.toBuffer)   *//定义一个长度为3的定长数组* **val** arr3 = *Array*(**"hadoop"**, **"storm"**, **"spark"**)  *//使用()来访问元素  println*(arr3(2))   *//////////////////////////////////////////////////  //变长数组（数组缓冲）  //如果想使用数组缓冲，需要导入import scala.collection.mutable.ArrayBuffer包* **val** ab = ArrayBuffer[Int](1,3,4)  **val** ab = new ArrayBuffer[Int]()  *//向数组缓冲的尾部追加一个元素  //+=尾部追加元素* ab += 1  *//追加多个元素* ab += (2, 3, 4, 5)  ab -= (3, 4)  *//追加一个数组++=* ab ++= *Array*(6, 7)  *//追加一个数组缓冲* ab ++= ArrayBuffer(8,9)  *//减少一个数组++=*  ab --= *Array*(6, 7) *//在数组某个位置插入元素用insert,第一个参数为插入元素的位置，后面的可变参数为插入的元素* ab.insert(0, -1, 0)  *//删除数组某个位置的元素用remove，第一个参数为要删除的元素位置，第二个参数为删除几个元素* ab.remove(8, 2)  *println*(ab)  *// 清空* ab1.clear() println(ab1)  } } |

## 遍历数组

1.增强for循环

2.生成器 to或者until，0 until 10 包含0不包含10

如何倒序输出？？ reverse

****

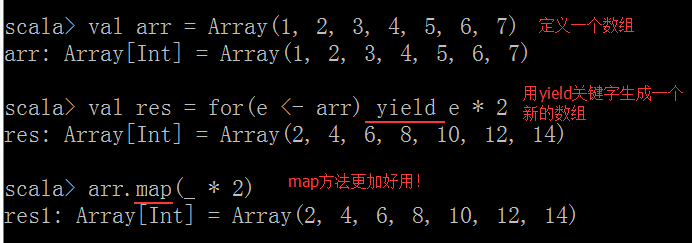
|  |
| --- |
| **object** ForArrayTest {   **def** main(args: Array[String]) {  *//初始化一个数组* **val** arr = *Array*(1,2,3,4,5,6,7,8)  *//增强for循环* **for**(i <- arr)  *println*(i)   *//好用的until会生成一个Range  //reverse是将前面生成的Range反转* **for**(i <- (0 until arr.length).reverse)  *println*(arr(i))  } }  **for**(i <- (arr.length-1 until （-1，-1）))  *println*(arr(i))  } |

## 数组转换

toArray 变长数组转换成定长数组

toBuffer 定长数组转换成变长数组

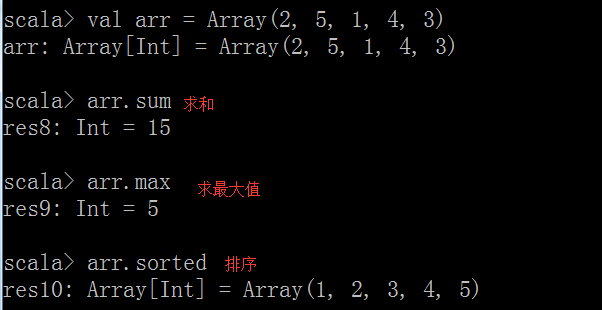
yield关键字将原始的数组进行转换会产生一个新的数组，原始的数组不变



|  |
| --- |
| **object** ArrayYieldTest {  **def** main(args: Array[String]) {  *//定义一个数组* **val** arr = *Array*(1, 2, 3, 4, 5, 6, 7, 8, 9)  *//将偶数取出乘以10后再生成一个新的数组* **val** res = **for** (e <- arr **if** e % 2 == 0) **yield** e \* 10  *println*(res.toBuffer)   *//更高级的写法,用着更爽  //filter是过滤，接收一个返回值为boolean的函数  //map相当于将数组中的每一个元素取出来，应用传进去的函数* **val** r = arr.filter(x => x % 2 == 0).map(x => x \* 10)  *println*(r.toBuffer)  } } |

## 数组常用方法

在Scala中，数组上的某些方法对数组进行相应的操作非常方便！



sorted默认是升序排序的。

arr. 然后输入Tab,可查看所有的方法。

val arr = Array(1,3,4,6,8)

数组反转：arr1.reverse

数组切片：arr.slice(1,4) 第一个起始位置，包含；第二个终止位置，不包含 [ )

# 元组Tuple

与数组或列表不同，元组可以容纳不同类型的对象，但它们也是不可变的。

**元组是不同类型元素的集合**

## 创建元组

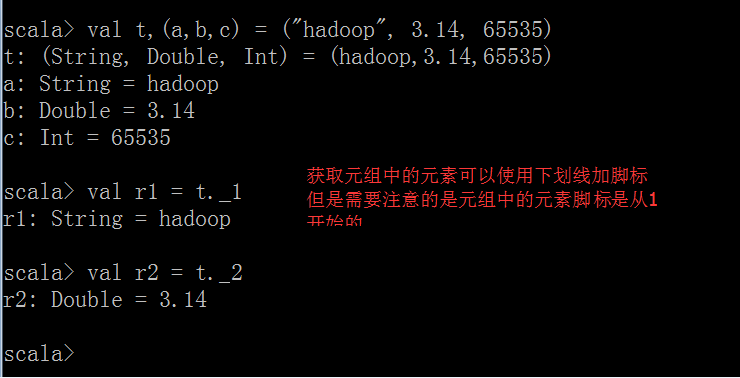
定义元组时，使用小括号将多个元素括起来，元素之间使用逗号分隔，元素的类型可以不同，元素的个数任意多个（不超过22个）

注意：元组没有可变和不可变之分，都是不可变的。

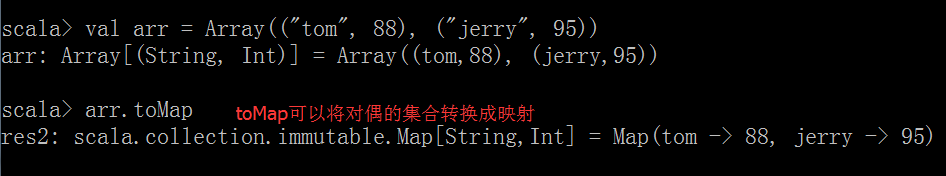
|  |
| --- |
| val t = (12.3, 1000, "spark")  val t1 = new Tuple1(1) // 必须1个元素  val t4 = new Tuple4(1,2.0,"",3) // 必须4个元素 |

## 获取元组中的值

获取元组的值使用下标获取，但是元组的下标时**从1开始**的，而不是0



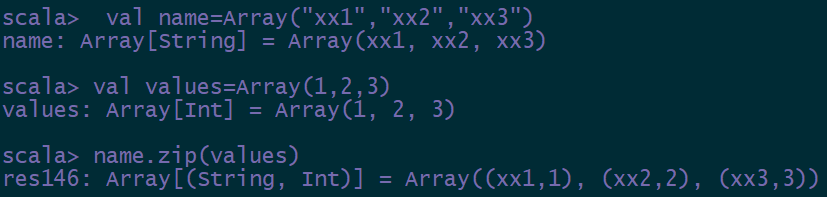
## 将对偶的集合转换成映射



## 拉链操作

**zip**命令可以将多个值绑定在一起，生成元组

|  |
| --- |
| val name=Array("xx1","xx2","xx3",”xx4”)  val values=Array(1,2,3)  name.zip(values) |



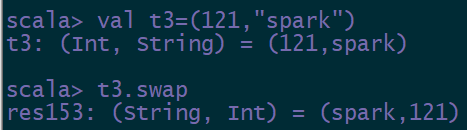
多个zip # name zip values zip values.map(\_\*10)

zipWithIndex 类似于zip,自带了索引，索引是从0 开始的。

注意：如果两个数组的元素个数不一致，拉链操作后生成的数组的长度为较小的那个数组的元素个数

## 元素交换

可以使用 Tuple.**swap** 方法来交换**对偶元组**的元素。



|  |
| --- |
| *// 定义元组* **var** *t* = (1, **"hello"**, **true**) *// 或者* **val** *tuple3* = **new** Tuple3(1, **"hello"**, **true**)  *// 访问tuple中的元素 println*(*t*.\_2) *// 访问元组总的第二个元素*  *// 对偶元组* **val** *tuple2* = (1, 3) *// 交换元组的元素位置, tuple2没有变化, 生成了新的元组* **val** *swap* = *tuple2*.swap |

元组类型Tuple1，Tuple2，Tuple3等等。目前在Scala中只能有22个上限，如果需要更多个元素，那么可以使用集合而不是元组。

练习1：定义一个方法或函数，求Double数组的平均值

练习2：定义一个方法或函数，求一个数组的最大值和最小值，返回两个最值组成的元组。

# 序列List

不可变list 长度不可变，内容也不可变 List

可变list 长度和内容都可变 ListBuffer

## 不可变序列

默认就是不可变序列，长度和内容都不可变

构造列表的两个基本单位是 **Nil**和 **::**

Nil 表示为一个空列表。

**创建List集合的两种方式：**

val list1= List[Int](1,2,3)

val list2 = 9::5::2::Nil

注意：:: 操作符是右结合的，该操作符就是将给定的头和尾创建一个新的列表

如9 :: 5 :: 2 :: Nil相当于 9 :: (5 :: (2 :: Nil))

注意：在Scala中列表要么为空,要么是一个head元素加上一个tail列表。

**列表的连接：**

可以使用++ 或 ::: 运算符或或 List.concat() 方法来连接两个或多个列表

**注意：对不可变List的所有的操作，全部生成新的List**

|  |
| --- |
| **object** ImmutListTest {   **def** main(args: Array[String]) {  *//创建一个不可变的集合* **val** lst1 = *List*(1,2,3)  *//将0插入到lst1的前面生成一个新的List* **val** lst2 = 0 :: lst1  **val** lst4 = 0 +: lst1    *//将一个元素添加到lst1的后面产生一个新的集合* **val** lst6 = lst1 :+ 3   **val** lst0 = *List*(4,5,6)  *//将2个list合并成一个新的List* **val** lst7 = lst1 ++ lst0  **val** lst8 = lst1 ++: lst0  **val** lst9 = lst1 ::: lst0  **val** lst10 = List.concat(lst1,lst0)  *// 列表反转*  lst1.reverse  // 列表头元素  lst1.head  // 列表的尾列表 lst1.tail  }  } |

**List和Array的对比：**

list不可变，表示长度不可变，内容也不可变

array 长度不可变，但是内容可变

不可变list和可变List的比较（List和ListBuffer）:

不可变List，长度不可变，值不可变

可变List,长度可变，值可变

## 可变序列

ListBuffer

需要显示导包 import scala.collection.mutable.\_

创建ListBuffer的两种方式：

**val** lb1 = ListBuffer[Int]() // 创建的同时可赋初值  
**val** lb2 = **new** ListBuffer[Int]() // 类型必须显示指定

添加元素：

+=方法和append方法，都可以添加多个元素。

|  |
| --- |
| **import** scala.collection.mutable.ListBuffer  **object** MutListTest {  **def** main(args: Array[String]) {  *//构建一个可变列表，初始有3个元素1,2,3* **val** *lst0* = ListBuffer[Int](1,2,3)  *//创建一个空的可变列表* **val** *lst1* = **new** ListBuffer[Int]  *//向lst1中追加元素，注意：没有生成新的集合  lst1* += （4,6）  *lst1*.append(5)  *//将lst0和lst1合并成一个新的ListBuffer 注意：生成了一个新集合* **val** *lst2* = *lst0* ++ *lst1*  *//将lst1中的元素追加到lst0中， 注意：没有生成新的集合  lst0* ++= *lst1   //将元素追加到lst0的后面生成一个新的集合* **val** *lst3* = *lst0* :+ 5  *//将元素追加到lst0的前面*  **val** *lst4* = 5 +: *lst0*  *// 去除元素* lb2 -= (1,3) lb2 --= *List*(7,9)  lb2.remove(1,2) *//去除元素 第一个参数是下标，第二个参数的个数*  *// 判断集合是否为空*  Lb2.isEmpty() }} |

List的转换：

可变list可以通过toList方法，生成新的List

list可以通过toBuffer，转变成ArrayBuffer

# 映射Map

在Scala中，把哈希表这种数据结构叫做映射

映射是K/V对 类型的值。

可变的map，还是不可变的map，都是Map

## 定义map

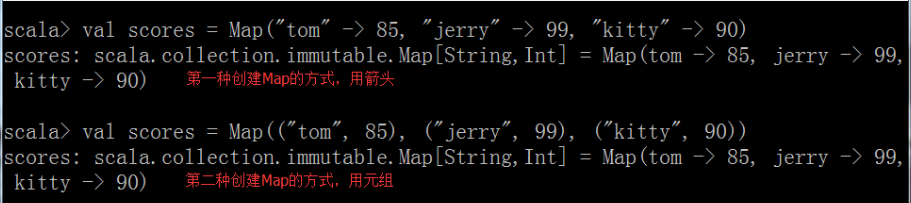
在Scala中，有两种Map，一个是immutable包下的Map，该Map中的内容不可变；另一个是mutable包下的Map，该Map中的内容可变。

默认是immutable包下的map，

|  |
| --- |
| *// 默认是immtable包下的Map*  **val** mp1 = *Map*((**"a"**,1),(**"b"**,2))**val** mp2 = *Map*(**"a"**->1,**"b"**->2) *// 添加元素之后生成新的map* **val** mp3 = mp2+(**"c"**->1) |

可使用mutable.Map

|  |
| --- |
| *// 导包* **import** scala.collection.mutable *// 创建集合* **val** mp4 = **new** mutable.HashMap[String,Int]() **val** mp5 = mutable.Map[String,Int]() |



## 添加元素

针对可变集合，多种赋值方式

|  |
| --- |
| mp4 += (**"e"**->8, **"f"**->9) mp4+= ((**"b1"**,121)) mp4.put(**"lyf"**,21) mp4(**"nvshen"**)=18 |

## 获取映射值

判断key是否存在 contains

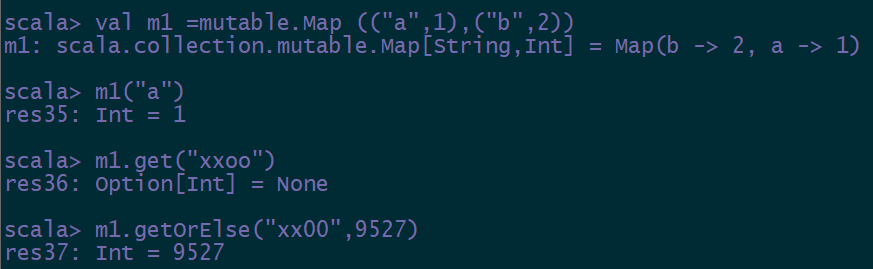
mp4.contains(“b”)

mp4(“a”)

mp4.get(“xxoo”)

如果没有值，赋予默认值：

mp4.getOrElse(“xxoo”,9527)



## 赋值和修改值

m1.getOrElse("d", 0)

m1("b") = 22

m1.updated("b",22) // 如果是不可变的Map,那么会生成一个新的map集合

## 删除元素

根据key来删除一个或多个值

m1 -= ("a")

m1.remove("a")

去除多个key：

m1 -= ("a","b")

m1 --= List(“key”)

## map遍历

for(i <- mp) println(i)

for((k,v) <- mp){println(k)}

for((**\_**,v) <- mp){println(v)}

**\_**是占位符，如果只需要遍历value，不需要遍历key，就可以用占位符

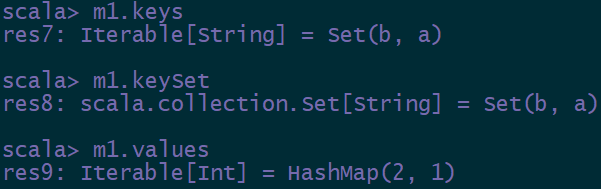
**交换k,v**

for((k,v) <- mp) yield (v,k)

mp.map(x=>(x.\_2,x.\_1))

mp.map(x=>x.swap)

## 获得keys和values



**合并**

使用 **++** 运算符或 **mp.++()** 方法来连接两个 Map，Map 合并时会移除重复的 key。

|  |
| --- |
| *// 合并时，相同的key元素会被覆盖* **val** colors1 = *Map*(**"nvshen"** ->18,**"nanshen"** -> 35) **val** colors2 = *Map*(**"bq"** -> 40,**"nvshen"** -> 20) **var** colors = colors1 ++ colors2 |

如果需要用到可变的Map和不可变的Map.

不导入具体的类，而是导入包名，在使用的时候，直接使用包名. 类

|  |
| --- |
| *// import scala.collection.mutable.Map*  **import** scala.collection.mutable  **val** mp2 = mutable.Map(**"a"** -> 1) |

# map|flatten|flatMap|foreach方法的使用

map 迭代集合中的每一个元素，然后按照自己的逻辑来修改元素

map方法操作集合中的每一个元素，返回值类型。

map方法有一个参数，是函数类型，该函数作用于集合的每一个元素上，对集合的每一个元素执行操作。有返回值。

map方法的内层返回值，是由函数的返回值决定的。

flatten 压平 如果有嵌套集合，就会把内层的嵌套去掉了

flatMap 先map 再flatten 先map 再压平

foreach 对每一个元素执行操作，相当于遍历 println 返回值是Unit

filter 过滤

map和foreach的区别：

1，底层实现不同，map方法利用隐式转换实现的Builder,foreach底层使用的是迭代器。

2，返回值不同，map返回值类型一般由函数返回值类型决定(map方法的参数就是一个函数)，foreach返回值为Unit

怎么选择？如果要求有返回值，就用map，如果是打印或者没有返回值的要求，使用foreach。

|  |
| --- |
| *// 定义一个数组* **val** *array* = *Array*[Int](2,4,6,9,3) *// map方法是将array数组中的每个元素进行某种映射操作, (x: Int) => x \* 2 为一个匿名函数, x 就是array中的每个元素* **val** *y* = *array* map((x: Int) => x \* 2) *// 或者这样写, 编译器会自动推测x的数据类型* **val** *z* = *array*.map(x => x\*2) *// 亦或者, \_ 表示入参, 表示数组中的每个元素值* **val** *x* = *array*.map(\_ \* 2)  *println*(*x*.toBuffer)   *println*(**"--------分割线--------"**)  *// 定义一个数组* **val** *words* = *Array*(**"hello tom hello jim hello jerry"**, **"hello Hatano"**)  *// 将数组中的每个元素进行分割 // Array(Array(hello, tom, hello, jim, hello, jerry), Array(hello, Hatano))* **val** *splitWords*: Array[Array[String]] = *words*.map(wd => wd.split(**" "**))  *// 此时数组中的每个元素经过split之后变成了Array, flatten是对splitWords里面的元素进行扁平化操作 // Array(hello, tom, hello, jim, hello, jerry, hello, Hatano)* **val** *flattenWords* = *splitWords*.flatten   *// 上述的2步操作, 可以等价于flatMap, 意味先map操作后进行flatten操作* **val** *result*: Array[String] = *words*.flatMap(wd => wd.split(**" "**))  *// 遍历数组, 打印每个元素 result*.foreach(*println*) |

# 案例wordCount

|  |
| --- |
| *// 定义一个数组* **val** *words* = *Array*(**"hello tom hello star hello sheep"**, **"hello tao hello tom"**)  *words*.flatMap(\_.split(**" "**)) *// 对数组中的每个元素进行切分, 并进行扁平化操作* .map((\_, 1)) *// 将数组的每个元素转换成一个对偶元组, 元组的第二个元素为1* .groupBy(\_.\_1) *// 对集合中的所有元素进行按单词分组, 相同单词的元组分到一组* .mapValues(\_.length) *// 对每个key 的value集合进行求长度操作* .toList *// 将map 转换成List  // 实现方式二 words*.flatMap(\_.split(**" "**)).groupBy(x => x).map(t => (t.\_1, t.\_2.length)).toList |

今日总结：

1. 函数和方法之间的区别和联系，特别是函数作为方法的参数。
2. 基本的集合框架。数组，元组，List，Map
3. 集合的常用方法 map flatMap flatten filter foreach
4. wordcount

# Set

不可变Set,可变Set

## 不可变的Set

长度和值都不可变，set中的元素不能重复

|  |
| --- |
| **object** ImmutSetTest {  **def** main(args: Array[String]) {  *// 默认是immtable包下的Set* **val** set1 = *Set* (1,4,6) *// 执行添加,删除的操作，都是生成了新的Set集合* **val** s2: Set[Int] = set1 + (10,12) set1 - (1) *println*(set1)  // 查看set集合内容  set1.foreach(println)   *//set中元素不能重复* **val** *set3* = *set1* ++ *Set*(5, 6, 7)  **val** *set0* = *Set*(1,3,4) ++ *set1* }  } |

## 可变的Set

可变Set中，remove方法，移除的是**元素**，而不是下标

ListBuffer中，remove方法，参数是下标

|  |
| --- |
| **import** scala.collection.mutable  **object** MutSetTest {  **def** main(args: Array[String]) {  **import** scala.collection.mutable.Set *// 可以在任何地方引入 可变集合* **val** mSet = **new** mutable.HashSet[Int]()  **val** mutableSet = Set(1,2,3)   mutableSet.add(4) *// mutableSet += 5* mutableSet += (5,15)  *// 添加set集合* mutableSet ++= Set(12,14) *// mutableSet -= 4* mutableSet -= (4,2)   *// remove方法，删除的不是下标，而是元素* mutableSet.remove(2)  *println*(mutableSet)  *// 转换为不可变集合* **val** another = mutableSet.toSet  *println*(another.getClass.getName) *// scala.collection.immutable.Set* }  } |

# Map和Option

mp.get(key)的返回值类型是Option[value的类型]

Option 类型，表示，值可能有，也可能没有

Option抽象类有两个子类，

一个是None,单例的，表示key不存在

一个是Some,多例的。Some(value) Some(value).get

在Scala中Option类型样例类用来表示可能存在或也可能不存在的值(Option的子类有Some和None)。Some包装了某个值，None表示没有值。

|  |
| --- |
| *// Option是Some和None的父类 // Some代表有（多例），样例类 // None代表没有（单例），样例对象* **val** mp = *Map*(**"a"** -> 1, **"b"** -> 2, **"c"** -> 3) **val** r: Int = mp(**"d"**)  *// Map 的get方法返回的为Option, 也就意味着 rv 可能取到也有可能没取到* **val** rv: Option[Int] = mp.get(**"d"**)  *// 如果rv=None时， 会出现异常情况* **val** r1 = rv.get  *// 使用getOrElse方法， // 第一个参数为要获取的key, // 第二个参数为默认值， 如果没有获取到key对应的值就返回默认值* **val** r2 = mp.getOrElse(**"d"**, -1)  *println*(r2) |

# 集合常用的方法

map

filter

过滤出满足条件的**所有**元素，并返回一个集合

filterNot

过滤出不满足条件的**所有**元素，并返回一个集合

find

过滤出满足条件的一个元素，并返回一个Option

如果说有结果值，返回值的是Some(元素值)，通过get方法来获取值

sorted

按元素的升序排序

sortBy

按照指定的条件排序

sortWith

接收两个参数，并进行比较

mapValues

类似于map，只是处理的是k-v类型中的v值，只能作用于map类型集合

1. 只能操作map集合
2. 作用于map集合中的values，然后返回map集合，也就是说map中的key是不变的

groupBy

按照指定条件分组

grouped grouped(size: Int): Iterator

按照指定元素个数进行分组

count **def** count(p: A => Boolean): Int

统计满足条件的元素个数

A: 取决于元素的类型。 Array[(String,Int)] 🡪 A： (String,Int) Array[Int] 🡪 A： Int

reduce \*\*\*\*\*

元素归并

参数是一个函数，这个函数有两个参数 累加值 元素值 调用的就是reduceLeft

|  |
| --- |
| val arr=Array("aa","bb","cc","dd")  arr.reduce((x,y)=>自定义操作)  自定义操作的方法，必须是x数据类型上支持的方法。x可以是任意类型  arr.reduce(\_ + \_)  arr.reduce(\_ ++ \_)  val lst = List(List("a"),List("b"),List("c"))  lst.reduce(\_++\_)  lst.reduce(\_:::\_) |

reduceLeft reduceRight

reduce底层调用的就是reduceLeft,只不过，reduce要求函数的输入类型和返回值类型必须一致，而reduceLeft,可以不一致。

|  |
| --- |
| val arr = Array(1,3,5,7,9)  reduce(\_ - \_) reduce((a,b)=>a-b)  false ---> true  a：累加值 b:元素值  1 1  3 1 3 1-3 -2  5 -2 5 -2 - 5 -7  7 -7 7 -7-7 -14  9 -14 9 -14 - 9 -23    (((((1)-3)-5)-7)-9) -23  reduce reduceLeft  (1-(3-(5-(7-(9))))) 5 |

reduce -🡪 reduceLeft

区别： reduce的2个输入参数类型，和输出值类型必须一致，

reduceLeft，可以支持两个不同类型的输入参数。

|  |
| --- |
| reduce[A1 >: A](op: (A1, A1) => A1)  reduceLeft[B >: A](op: (B, A) => B) |

reduce

fold foldLeft foldRight

|  |
| --- |
| **def** fold[A1 >: A](z: A1)(op: (A1, A1) => A1): A1  **def** foldLeft[B](z: B)(op: (B, A) => B): B |

fold(0)(\_ + \_)

fold有两个参数，第一个参数是默认值，第二个参数是一个函数，该函数有两个参数 累加值 元素值 调用的就是foldLeft

fold 要求函数的2个输入参数类型必须一致，foldLeft 可以允许函数的2个输入参数类型

不一致

fold foldLeft foldRight 相比于 reduce reduceLeft reduceRight ，多了一个初始值参数运算。其他的都一样，+ ++ 取决于 元素的类型。

|  |
| --- |
| 示例：求平均值  **val** d1 = *Array*((**"bj"**,28.1), (**"sh"**,28.7), (**"gz"**,32.0), (**"sz"**, 33.1)) **val** d2 = *Array*((**"bj"**,27.3), (**"sh"**,30.1), (**"gz"**,33.3)) **val** d3 = *Array*((**"bj"**,28.2), (**"sh"**,29.1), (**"gz"**,32.0), (**"sz"**, 30.5))  *// 1,需要把数据组装到一起* **val** data1: Array[(String, Double)] = d1.union(d2).union(d3) *// d1 union d2 union d3  //d1 ++ d2 ++ d3  // 2 分组 按照城市名称来分组* **val** data2: Map[String, Array[(String, Double)]] = data1.groupBy(t=>t.\_1)  *// 统计* **val** data4 = data2.mapValues({  kv=>  *// kv数据类型： Array[(String,Double)]  // t的数据类型是元组（String,Double）*  }) *println*(**"--------111------------"**) *println*(data4)   *// 3 统计 拿到这几个月份的温度的总值，然后再求平均* **val** result: Map[String, Double] = data2.map {  t =>  *// t (String,Array[(String, Double)])* **val** city = t.\_1  *// foldLeft 第一个是累加值 第二个是元素值* **val** wendu: Double = t.\_2.foldLeft(0d)({  *// total是Double a是(String, Double)* (total, a) =>  total + a.\_2  }) / t.\_2.length  (city, wendu) } *println*(result) |

交集，并集 差集 U

intersect union diff

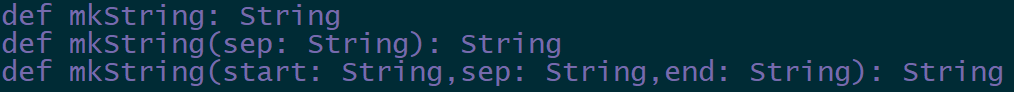
**union**是一个轻量级的方法，union没有去重

//比较常用的方法：先union，然后再分组

distinct 元素去重

mkString 把集合中的所有元素拼接成**字符串**

mkString(分隔符)



arr1.reduce(\_+"-" + \_)

arr1.mkString("-")

take(n) 获取集合中前几个元素 没有排序

slice(from,until) 截取元素，提取元素列表中的from 到until位置的元素

根据角标获取元素的值， [ ）

聚合 aggregate

**val** *arr* = *List*(*List*(1, 2, 3), *List*(2))

**val** *result* = *arr*.aggregate(0)(\_+\_.sum, \_+\_)

# 练习：

1. 计算一个Array[Double]数组的平均值

arr.sum / arr.length

1. 编写一个函数或方法，返回数组中最小值和最大值的对偶 （元组）

val f = (arr:Array[Double]) => (arr.min, arr.max)

f(arr)

1. 编写一个函数或方法，getValues(values: Array[lnt]，v：Int)，返回数组中小于v、等于v和大于v的元素个数，要求三个值一起返回

val getValues = (values:Array[Int], v:Int) => (values.count(\_ > v), values.count(\_ < v), values.count(\_ == v))

val a = Array(1,2,3,45,6,32)

getValues(a, 3)

1. 数组反转，两两交换 Array(1,2,3,5,6,7,9) 2 1 5 3 7 6 9 to until 步长

val a = Array(1,2,3,5,6,7,9)

val tmp = a.grouped(2)

val newA = new ArrayBuffer[Int]()

while(tmp.hasNext){

newA ++= tmp.next().toList.reverse

}

**def** arrSwap(arr:Array[Int]) = {  
 **for**(i <- 0 until arr.length-1 **if** i%2 == 0){  
 **var** temp = arr(i)  
 arr(i) = arr(i+1)  
 arr(i+1) = temp  
 }  
 *//返回值，不能少* arr  
}

**def** arrSwap2(arr:Array[Int]) = {  
 **for**(i <- 0 until (arr.length-1, 2)){  
 **var** temp = arr(i)  
 arr(i) = arr(i+1)  
 arr(i+1) = temp  
 }  
 *//返回值，不能少* arr  
}

**val** arr = Array[Int](2, 5, 9, 8, 6)  
  
**val** grouped = arr.grouped(2)  
**val** map = grouped.flatMap(t => t.reverse)  
*println*(map.toList)

1. 直接数组的整体反转
2. 求平均值

|  |
| --- |
| **object** ListTest{   **def** main(args: Array[String]): Unit = {  **val** d1 = *Array*((**"bj"**, 28.1), (**"sh"**, 30.3), (**"sz"**, 39.2), (**"gz"**, 29.8))  **val** d2 = *Array*((**"bj"**, 28.1), (**"sh"**, 30.3), (**"sz"**, 39.2))  **val** d3 = *Array*((**"bj"**, 27.6), (**"sh"**, 20.3), (**"sz"**, 38.2), (**"gz"**, 28.8))  **val** data = d1 ++ d2 ++ d3  **val** grouped = data.groupBy(\_.\_1)  **val** result = grouped.map(t => {  **val** sum = t.\_2.map(\_.\_2).sum  **val** len = t.\_2.length  (t.\_1, sum / len)  })  result.foreach(*println*)  } }  } } |

6，key value互换

val lst = List("Id1-The Spark","Id2-The Hadoop","Id3-The Spark")

结果值：

The-Id1 Id2 Id3

Spark-Id1 Id3

Hadoop-Id2

写思路 ，再用代码实现每一步的思路

|  |
| --- |
| **object** ListTest{   **def** main(args: Array[String]): Unit = {  **val** lst = *List*(**"Id1-The Spark"**,**"Id2-The Hadoop"**,**"Id3-The Spark"**)  **val** result = lst.map(t => {  **val** key = t.split(**"-"**)(0)  **val** value = t.split(**"-"**)(1)  **val** values = value.split(**" "**)   values.map(t => {  (t, key)  })  })   **val** tmp = result.flatten  **val** grouped = tmp.groupBy(\_.\_1)  **val** t = grouped.map(t => {  **val** key = t.\_1  **val** value = t.\_2.map(\_.\_2)  (key, value)})  **val** r = t.map(e => {  e.\_1 + **"-"** + e.\_2.reduce(\_+**" "**+\_)})   r.foreach(*println*)   } } |